Improved Time-Space Trade-Offs for Computing Voronoi Diagrams*

Bahareh Banyassady[1], Matias Korman[2], Wolfgang Mulzer[3], André van Renssen[4], Marcel Roeloffzen[5], Paul Seiferth[6], and Yannik Stein[7]

1 **Institut für Informatik, Freie Universität Berlin, Berlin, Germany**
bahareh@inf.fu-berlin.de
2 **Tohoku University, Sendai, Japan**
mati@dais.is.tohoku.ac.jp
3 **Institut für Informatik, Freie Universität Berlin, Berlin, Germany**
mulzer@inf.fu-berlin.de
4 **National Institute of Informatics (NII), Tokyo, Japan; and JST, ERATO, Kawarabayashi Large Graph Project, Tokyo, Japan**
andre@nii.ac.jp
5 **National Institute of Informatics (NII), Tokyo, Japan; and JST, ERATO, Kawarabayashi Large Graph Project, Tokyo, Japan**
marcel@nii.ac.jp
6 **Institut für Informatik, Freie Universität Berlin, Berlin, Germany**
pseiferth@inf.fu-berlin.de
7 **Institut für Informatik, Freie Universität Berlin, Berlin, Germany**
yannikstein@inf.fu-berlin.de

Abstract

Let P be a planar n-point set in general position. For $k \in \{1, \ldots, n-1\}$, the Voronoi diagram *of order* k is obtained by subdividing the plane into *regions* such that points in the same cell have the same set of nearest k neighbors in P. The *(nearest point) Voronoi diagram* (NVD) and the *farthest point Voronoi diagram* (FVD) are the particular cases of $k = 1$ and $k = n-1$, respectively. It is known that the family of all higher-order Voronoi diagrams of order 1 to K for P can be computed in total time $O(nK^2 + n \log n)$ using $O(K^2(n-K))$ space. Also NVD and FVD can be computed in $O(n \log n)$ time using $O(n)$ space.

For $s \in \{1, \ldots, n\}$, an s-workspace algorithm has random access to a read-only array with the sites of P in arbitrary order. Additionally, the algorithm may use $O(s)$ words of $\Theta(\log n)$ bits each for reading and writing intermediate data. The output can be written only once and cannot be accessed afterwards.

We describe a deterministic s-workspace algorithm for computing an NVD and also an FVD for P that runs in $O((n^2/s) \log s)$ time. Moreover, we generalize our s-workspace algorithm for computing the family of all higher-order Voronoi diagrams of P up to order $K \in O(\sqrt{s})$ in total time $O\big(\frac{n^2K^6}{s} \log^{1+\varepsilon} K \cdot (\log s / \log K)^{O(1)}\big)$, for any fixed $\varepsilon > 0$. Previously, for Voronoi diagrams, the only known s-workspace algorithm was to find an NVD for P in *expected* time $O((n^2/s) \log s + n \log s \log^* s)$. Unlike the previous algorithm, our new method is very simple and does not rely on advanced data structures or random sampling techniques.

1998 ACM Subject Classification F.2.2 [Nonnumerical Algorithms and Problems] Geometrical Problems and Computations

* MK was supported in part by the ELC project (MEXT KAKENHI No. 12H00855 and 15H02665). BB, WM and PS were supported in part by DFG Grants MU 3501/1 and MU 3501/2. YS was supported by the DFG within the research training group "Methods for Discrete Structures" (GRK 1408) and by GIF Grant 1161.

34th Symposium on Theoretical Aspects of Computer Science (STACS 2017).
Editors: Heribert Vollmer and Brigitte Vallée; Article No. 9; pp. 9:1–9:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases memory-constrained model, Voronoi diagram, time-space trade-off

Digital Object Identifier 10.4230/LIPIcs.STACS.2017.9

1 Introduction

In recent years, we have seen an explosive growth of small distributed devices such as tracking devices and wireless sensors. These devices are small, have only limited energy supply, are easily moved, and should not be too expensive. To accommodate these needs, the amount of memory on them is tightly budgeted. This poses a significant challenge to software developers and algorithm designers: how to create useful and efficient programs in the presence of strong memory constraints?

Memory constraints have been studied since the introduction of computers (see for example Pohl [27]). The first computers often had limited memory compared to the available processing power. As hardware progressed this gap became smaller, other concerns became more important, and the focus of algorithms research shifted away from memory-constrained models. Memory constraints are again an important problem to tackle with these new devices as well as huge datasets available through cloud computing.

An easy way to model algorithms with memory constraints is to assume that the input is stored in a read-only memory. This is appealing for several reasons. From a practical viewpoint, writing to external memory is often a costly operation, e.g., if the data resides on a read-only medium such as a DVD or on hardware where writing is slow and wears out the hardware, such as flash memory. Similarly, in concurrent environments, writing operations may lead to race conditions. Thus, from a practical viewpoint, it is useful to limit or simply disallow writing operations. From a theoretical viewpoint, this model is also advantageous: keeping the working memory separate from the (read-only) input memory allows for a more detailed accounting of the space requirements of an algorithm and for a better understanding of the required resources. In fact, this is exactly the approach taken by computational complexity theory to define complexity classes that model *sublinear* space requirements, such as the complexity class of problems that use logarithmic amount of memory space [3].

Some of the earliest results in this setting concern the sorting problem [24, 25]. Suppose we want to sort data items whose total description complexity is n bits, all of them residing in a read-only memory. For our computations we can use a workspace of $O(b)$ bits freely (both read and write operations are allowed). Then it is known that the time-space product must be $\Omega(n^2)$ [13], and a matching upper bound for the case $b \in \Omega(\log n) \cap O(n/\log n)$ was given by Pagter and Rauhe [26] (b is the available workspace in bits). A result along these lines is known as a *time-space trade-off* [28].

The model used in this work was introduced by Asano *et al.* [6], following similar earlier models [14, 17]. Asano *et al.*provided constant workspace algorithms for many classic problems from computational geometry, such as computing convex hulls, Delaunay triangulations, Euclidean minimum spanning trees, or shortest paths in polygons [6]. Since then, the model has enjoyed increasing popularity, with work on shortest paths in trees [7] and time-space trade-offs for computing shortest paths [4, 20], visibility regions in simple polygons [9, 11], planar convex hulls [10, 18], general plane-sweep algorithms [19], or triangulating simple polygons [4, 5, 2]. We refer the reader to [21] for a deeper survey of the latest results.

Let us specify our model more precisely: we are given a planar point set P of n points stored in a read-only array that allows random access. Furthermore, we may use $O(s)$ variables (for a parameter $s \in \{1, \ldots, n\}$) for reading and writing. We assume that all the

data items and pointers are represented by $\Theta(\log n)$ bits. Other than this, the model allows the usual word RAM operations.

We consider the problem of computing various Voronoi diagrams for P, namely the *nearest point Voronoi diagram* NVD(P), the *furthest point Vornoi diagram* FVD(P), and the family of *higher-order Voronoi diagrams* up to a given order $K \in O(\sqrt{s})$. In most workspaces, the output cannot be stored explicitly. Thus, we require that the algorithm reports the edges of the Voronoi diagrams one-by-one in a write-only data structure (once written, they cannot be read or further modified). Note that we may report edges of the Voronoi diagrams in any order, but we are not allowed to report an edge more than once.

Previous Work. If we forego memory constraints, it is well known that both NVD(P) and FVD(P) can be computed in $O(n \log n)$ time using $O(n)$ space [8, 12]. For computing a single order-k Voronoi diagram, the best randomized algorithm takes $O(n \log n + nk \log k)$ expected time [16] using $O(nk)$ space, and the best deterministic algorithm takes $O(nk \log^{1+\varepsilon} k \cdot (\log n / \log k)^{O(1)})$ time [15] and $O(nk)$ space, for $\varepsilon > 0$ (as usual, the big O notation hides multiplicative factors that depend on ε). The family of all higher-order Voronoi diagrams up to order K can be computed in $O(nK^2 + n \log n)$ time using $O(K^2(n-K))$ space [1, 23].

Very few memory-constrained algorithms that compute Voronoi diagrams exist in the literature. Asano *et al.* [6] showed that NVD(P) can be found in $O(n^2)$ time in a $O(1)$ workspace. Korman *et al.* [22] gave a time-space trade-off for computing Voronoi diagrams. Their algorithm is based on random sampling and achieves an expected running time of $O((n^2/s) \log s + n \log s \log^* s))$ using $O(s)$ words of workspace.

Results. In this paper we introduce a time-space trade-off algorithm that improves these results, and gives a simpler and more flexible approach to obtain the diagrams. In Section 3 we show that the approach of Asano *et al.* [6] can be used to compute FVD(P). In Section 4 we introduce a new time-space trade-off for computing NVD(P) and FVD(P). Unlike the approach of Korman *et al.* [22], this new algorithm is deterministic, and slightly faster (it runs in $O((n^2/s) \log s)$ time using $O(s)$ words of workspace).

Finally, in Section 5, we use the s-workspace algorithm as a base in a novel pipelined fashion to compute the family of all Voronoi diagrams of order 1 to $K \in O(\sqrt{s})$ in total time $O\big(\frac{n^2K^6}{s} \log^{1+\varepsilon} K \cdot (\log s / \log K)^{O(1)}\big)$, for $\varepsilon > 0$, using $O(s)$ words of workspace. The main idea is to compute edges of the different Voronoi diagrams simultaneously. To compute the edges of a diagram we use edges of the previous order Voronoi diagram. However, this needs to be coordinated carefully, in order to prevent edges from being reported multiple times.

2 Preliminaries and Notation

Throughout the paper we denote by $P = \{p_1, \ldots, p_n\}$ a set of $n \geq 3$ *sites* in the plane. We assume general position, which here means that no three sites of P lie on a common line and no four sites of P lie on a common circle. The *nearest point Voronoi diagram* of P, NVD(P), is obtained by classifying the points in the plane according to their nearest neighbors in P. To define our terminology, we recall some classic and well-known properties of NVD(P) [8, 12]. For each site $p \in P$, the open set of points in $\mathbb{R}^2$ that have p as their unique nearest neighbor in P is called the *Voronoi cell* of p. Each *Voronoi edge* between two points $p, q \in P$ consists of all points in the plane with p, q as their only two nearest neighbors. Whenever it exists, the Voronoi edge is a subset of the bisector $B(p, q)$ of p, q defined as the line containing all points that are equidistant to p and q. Note that our general position

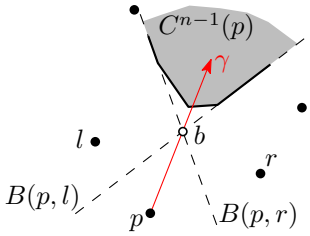

Figure 1 An illustration of Fact 3.2: if p is on $\text{conv}(P)$, we can find a ray that intersects the boundary of $C^{n-1}(p)$.

assumption together with $n \geq 3$ guarantees that each Voronoi edge is an open segment or halfline. Finally, *Voronoi vertices* are the points in the plane that have exactly three nearest neighbors in P. By general position, every point in $\mathbb{R}^2$ is either a Voronoi vertex, or it lies on a Voronoi edge or in a Voronoi cell. The Voronoi vertices and the Voronoi edges form the set of vertices and edges of a plane graph whose faces are the Voronoi cells. The complexity of this graph is $O(n)$.

The *farthest point Voronoi diagram* of P, $\text{FVD}(P)$, is defined analogously. Farthest Voronoi cells, edges, and vertices are obtained by replacing the term "nearest neighbor" by the term "farthest neighbor" in the respective definitions. Again, the farthest Voronoi vertices and edges constitute the vertices and edges of a plane graph of complexity $O(n)$. However, unlike in $\text{NVD}(P)$, in $\text{FVD}(P)$ it is not necessarily the case that all sites in P have a corresponding cell in $\text{FVD}(P)$. In fact, the sites with non-empty farthest Voronoi cells correspond exactly to the sites on the *convex hull* of P, $\text{conv}(P)$. Furthermore, all the cells in $\text{FVD}(P)$ are unbounded, and hence $\text{FVD}(P)$, considered as a plane graph, is a tree.

Now for $k \in \{1, \ldots, n-1\}$, the Voronoi diagram *of order* k is obtained by classifying the points in the plane according to the *set* of their nearest k neighbors in P. We denote the k-order Voronoi diagram of P by $\text{VD}^k(P)$. Observe that $\text{NVD}(P) = \text{VD}^1(P)$ and $\text{FVD}(P) = \text{VD}^{n-1}(P)$. For each subset $Q \subset P$ of k sites from P, we denote the Voronoi cell of order k of Q by $C^k(Q)$. We know that $\text{VD}^k(P)$ is a plane graph of complexity $O(k(n-k))$ [8, 23]. For simplicity, the Voronoi cell of $p \in P$ in $\text{NVD}(P)$ and $\text{FVD}(P)$ are denoted respectively by $C^1(p)$ and $C^{n-1}(p)$.

3 A Constant Workspace Algorithm for FVDs and NVDs

We are given a planar n-point set $P = \{p_1, \ldots, p_n\}$ in a read-only array, and our task is to report the edges of $\text{NVD}(P)$ and of $\text{FVD}(P)$ using only a constant amount of additional workspace. First, we show how to find a single edge of a cell of $\text{NVD}(P)$ or of $\text{FVD}(P)$. Then, we extend this approach to find all the edges of $\text{NVD}(P)$ and $\text{FVD}(P)$. We summarize the properties of $\text{FVD}(P)$, that are required by our algorithms, in the following two facts. See, e.g., the book by Aurenhammer, Klein, and Lee [8] for more details.

▶ **Fact 3.1.** *Let P be a planar n-point set in general position and $p \in P$. The cell $C^{n-1}(p)$ is empty if and only if p is in the interior of the convex hull of P. If p is on the convex hull of P and $r, l \in P$ are the two adjacent sites of p on $\text{conv}(P)$, then both a subset of $B(p,l)$ and a subset of $B(p,r)$ are unbounded edges of $C^{n-1}(p)$.*

▶ **Fact 3.2.** *Let P be a planar n-point set in general position. Let $p \in P$ be on $\text{conv}(P)$ and let $l, r \in P$ be its adjacent sites on $\text{conv}(P)$. Let b be the intersection point of $B(p,l)$ and*

$B(p,r)$. Then, the ray γ from p towards b intersects the boundary of $C^{n-1}(p)$ (not necessarily at b); see Figure 1.

▶ **Lemma 3.3.** *Let P be a planar n-point set in general position in a read-only array. For any $p \in P$, we can determine whether $C^{n-1}(p)$ is empty, in $O(n)$ time and constant workspace. Furthermore, if $C^{n-1}(p)$ is not empty, we can find a ray that intersects the boundary of $C^{n-1}(p)$ in the same time and space.*

Proof. By Fact 3.1, it suffices to check whether p lies inside $\text{conv}(P)$. This can be done using simple *gift-wrapping*: Pick an arbitrary site $q \in P \setminus \{p\}$. Scan through P and find the sites p_{cw} and p_{ccw} in P which make, respectively, the largest clockwise angle and the largest counterclockwise angle with the ray pq, such that both angles are at most π. Thus, p_{cw} and p_{ccw} are easily obtained in $O(n)$ time using constant workspace. If the angle $p_{\text{cw}}pp_{\text{ccw}}$ (that contains q) is larger than π, then p is inside $\text{conv}(P)$ and consequently $C^{n-1}(p)$ is empty. Otherwise, p is on $\text{conv}(P)$, and both p_{cw} and p_{ccw} are its adjacent sites on $\text{conv}(P)$. By Fact 3.2, the ray from p through $B(p,p_{\text{cw}}) \cap B(p,p_{\text{ccw}})$ intersects the boundary of $C^{n-1}(p)$. ◀

▶ **Lemma 3.4.** *Let P be a planar n-point set in general position in a read-only array. Suppose we are given a site $p \in P$ and a ray γ that emanates from p and intersects the boundary of $C^1(p)$ (or $C^{n-1}(p)$). Then, we can report the edge e of $C^1(p)$ (or $C^{n-1}(p)$) that intersects γ, in $O(n)$ time using $O(1)$ words of workspace.*

Proof. For all sites $p' \in P$, we consider the bisector $B(p,p')$. Among all these bisectors, we find the bisector ℓ_e that intersects γ closest to (farthest from) p. The edge e is a subset of ℓ_e. We can find ℓ_e by scanning the sites of P and storing the closest (farthest) bisector so far in each step. To find the portion of ℓ_e that forms a Voronoi edge in $\text{NVD}(P)$ (or $\text{FVD}(P)$), we do a second scan of P. For any $p' \in P$ we check where $B(p,p')$ intersects ℓ_e. Each such intersection removes a section from ℓ_e which cannot appear in $\text{NVD}(P)$ (or $\text{FVD}(P)$). From each infinite side of ℓ_e, there is at most one intersection that removes the biggest portion of ℓ_e and thus defines the endpoint of e from that side. Thus, in each step we store only the most restricted intersection from each side (if it exists). Overall, we can find the edge e of $C^1(p)$ (or of $C^{n-1}(p)$) in $O(n)$ time using $O(1)$ words of workspace. ◀

▶ **Theorem 3.5.** *Suppose we are given a planar n-point set $P = \{p_1, \ldots, p_n\}$ in general position in a read-only array. We can find all the edges of* $\text{NVD}(P)$ *(or of* $\text{FVD}(P)$*) in $O(n^2)$ time using $O(1)$ words of workspace.*

Proof. We restate the strategy that was previously used by Asano *et al.* [6] for $\text{NVD}(P)$. We give the details and show that a similar strategy works for $\text{FVD}(P)$.

We go through the sites in P one by one. In step i, we process $p_i \in P$ to detect all edges of $C^1(p_i)$ (or $C^{n-1}(p_i)$). To do this, we first need a ray γ to apply Lemma 3.4. For $\text{NVD}(P)$ we choose a ray γ from p_i to an arbitrary site of $P \setminus \{p_i\}$. In this way, we know that γ intersects the boundary of $C^1(p_i)$. For $\text{FVD}(P)$ first check if the Voronoi cell of p_i is non-empty. If so, we use Lemma 3.3 to find a ray γ that intersects the boundary of $C^{n-1}(p_i)$. From here on, the algorithms for enumerating the edges of $\text{NVD}(P)$ and $\text{FVD}(P)$ are similar. Having the ray γ at hand, we use Lemma 3.4 to find an edge e of $C^1(p_i)$ (or of $C^{n-1}(p_i)$). We consider the ray γ' from p_i to the left endpoint of e (if it exists), and we apply Lemma 3.4 to find the adjacent edge e' of e in $C^1(p_i)$ (or in $C^{n-1}(p_i)$). Note that the ray will now hit both e and e'. This can be fixed by making a symbolic perturbation to γ' so that only e' is hit. We proceed in a similar manner to find further edges of $C^1(p_i)$ (or $C^{n-1}(p_i)$) in counterclockwise direction. The process continues until we reach e again or until we find an

unbounded edge of $C^1(p_i)$ (or of $C^{n-1}(p_i)$). In the latter case, we start again from the right endpoint of e (if it exists), and we find the remaining edges of $C^1(p_i)$ (or of $C^{n-1}(p_i)$) in clockwise direction, stopping the process when the current edge is unbounded.

Using this process we detect each edge twice (i.e., edges that are a subset of $B(p_i, p_j)$ will be detected when processing p_i and p_j). To avoid reporting the same edge twice, when we find an edge e of $C^1(p_i)$ (or of $C^{n-1}(p_i)$) with $e \subseteq B(p_i, p_j)$ we report e if and only if $i < j$. Since NVD(P) (or FVD(P)) has $O(n)$ edges, and reporting one edge takes $O(n)$ time and $O(1)$ words of workspace, the result follows. ◀

4 Obtaining a Time-Space Trade-off

Now we adapt the previous algorithm to a time-space trade-off in which we have a workspace of $O(s)$ variables. As before, we are given a planar n-point set $P = \{p_1, \ldots, p_n\}$ in general position in a read-only array, and we would like to report the edges of NVD(P) or FVD(P) as quickly as possible. For this, we first show how to find one edge of s different cells of NVD(P) or FVD(P) simultaneously. After that, we describe how to coordinate these simultaneous searches to find all the edges of NVD(P) or FVD(P).

▶ **Lemma 4.1.** *Suppose we are given a set $V = \{v_1, \ldots, v_s\}$ of s sites in P, and for each $i = 1, \ldots, s$, a ray γ_i emanating from v_i such that γ_i intersects the boundary of $C^1(v_i)$ (or* FVD(P)*). Then we can report for each $i = 1, \ldots, s$, the edge e_i of $C^1(v_i)$ (or* FVD(P)*) that intersects γ_i, in $O(n \log s)$ total time using $O(s)$ words of workspace.*

Proof. For ease of reading we provide the proof only for NVD(P) (the proof for FVD(P) is obtained by simply replacing NVD(P) by FVD(P)). The algorithm has two phases. In the first phase, for $i = 1, \ldots, s$, we find the line ℓ_i that contains e_i, and in the second phase, for $i = 1, \ldots, s$, we find the portion of ℓ_i which is in NVD(P), i.e., we find the endpoints of e_i.

The first phase proceeds as follows: we select the first batch $Q_1 = \{p_1, \ldots, p_s\}$, of s sites of P, and we compute NVD($V \cup Q_1$). Since $V \cup Q_1$ has $O(s)$ sites, we can compute it in $O(s \log s)$ time using $O(s)$ workspace. Now, for $i = 1, \ldots s$, we find the edge $e'_i \in$ NVD($V \cup Q_1$) of the cell of v_i that intersects γ_i, we store the line spanned by e'_i in ℓ_i, and proceed to the next batch of s sites. In general in step j, for $j = 1, \ldots, n/s$, we select Q_j which is the j^{th} batch of s sites of P, and we compute NVD($V \cup Q_j$). Then, for $i = 1, \ldots, s$, we find the edge of the cell of v_i in NVD($V \cup Q_j$) that intersects γ_i. We update ℓ_i to the line spanned by this new edge only if it intersects γ_i closest to v_i.

We claim that after all n/s batches of P have been scanned, ℓ_i is the line that contains the edge of $C^1(v_i)$ that intersects γ_i. To see this, recall that the edge e_i in NVD(P) lies on a bisector between v_i and another site $p \in P \setminus \{v_i\}$. Thus, this line is among the lines considered in NVD($V \cup Q_j$), for $j = 1, \ldots, n/s$.

In the second phase, we again process P in batches of size s. In the first step, we take the first batch of s sites of P, $Q_1 = \{p_1, \ldots, p_s\}$, and we again compute NVD($V \cup Q_1$). For $i = 1, \ldots, s$, we find the portion of ℓ_i inside the cell of v_i in NVD($V \cup Q_1$), and we store it in e_i. In step j, for $j = 1, \ldots, n/s$, we select Q_j, the j^{th} batch of s sites of P, and we compute NVD($V \cup Q_j$). For $i = 1, \ldots, s$, we update the endpoints of e_i (the new e_i is simply the intersection of the previous e_i and the cell of v_i in NVD($V \cup Q_j$)). At the end of step j, the variable e_i represents the best candidate that we have found so far. In other words, e_i contains the portion of ℓ_i whose nearest site is v_i (among the sites $V \cup_{k=1}^{j} Q_k$). Further note that, due to the Voronoi properties, e_i is a connected subset of ℓ_i (that is, a ray or a segment). In particular, it can be described with its at most two endpoints. Thus, after n/s steps, e_i is the edge of $C^1(v_i)$ that intersects γ_i.

In each step of each phase, we construct a Voronoi diagram in $O(s \log s)$ time using $O(s)$ workspace. Since the total number of steps is n/s, the running time of the algorithm is $O(n \log s)$. At each step we store only $O(s)$ sites (and a constant amount of information on each site), so the space bounds are not exceeded. ◀

Now we can describe our algorithm. We repeatedly use Lemma 4.1 to find an edge of s different sites at once. Once all edges of a site have been found, it is discarded and we proceed to the next one. Since the Voronoi diagram has $O(n)$ edges and at each iteration we find s edges, after $O(n/s)$ steps, fewer than s sites will remain to be processed. At this step we stop using Lemma 4.1. We do so because if the number of edges remaining to be found for each site is unbalanced, we cannot afford to continue using Lemma 4.1 (each iteration will still cost $O(n \log s)$ to execute but $o(s)$ edges would be found). Instead, we will treat these remaining sites differently. We say that a site is *small* if all of its edges are found while using Lemma 4.1, and *big* otherwise. By the way in which our algorithm works, small sites have $O(n/s)$ edges in their Voronoi cell, but big ones may have many edges (if they are among the last sites to be processed they do not have necessarily many edges).

Our algorithm has three phases. In the first phase we process the whole input to detect which sites are the big ones (no edge will be reported in this phase). The second phase scans the input again and reports all edges that belong to a bisector between a small site and some other site. The third and final phase reports edges between two big sites.

First phase. Recall that the aim of this phase is to identify the big sites. We describe how we use Lemma 4.1 in more details. We want to scan all sites whose associated Voronoi cell is nonempty. For NVD(P), this is trivial since all sites have a nonempty cell in NVD(P). Hence, it suffices to scan them sequentially. The starting ray can be constructed in the same way as in Theorem 3.5. If we are interested in computing FVD(P) instead, we use the algorithm of Darwish and Elmasry [18]. This algorithm reports all sites that belong to the convex hull of P in $O(\frac{n^2}{s \log n})$ time using $O(s)$ words of workspace. Sites are reported one by one in clockwise order along the convex hull. Thus, we will use the output of the algorithm of Darwish and Elmasry as our input instead: we run their algorithm storing any sites that would be reported. Whenever we gathered s sites, we pause the execution of the convex hull computation and process those sites. Whenever more sites are needed, we simply resume the execution of the convex hull algorithm. Since sites are reported in clockwise order, whenever a site is reported we know both of its neighbors. This allows us to use Fact 3.2 to find a starting ray for each site.

Regardless of the order in which we process the sites, we keep s sites from P in memory. Now we apply Lemma 4.1 to compute, for every site in memory, one edge of its cell. Once the edge is computed, as in Theorem 3.5, we update the rays to look for the next edge of each cell. Whenever all edges of a cell have been found we remove the corresponding site from memory, and we insert the next site from P into the working memory. Since the Voronoi diagram has $O(n)$ edges, and at each iteration we find s edges, after $O(n/s)$ steps, fewer than s sites remain in memory, and all the other sites of P have been processed.

When the first phase of the algorithm ends, we identified the big sites (those remaining to be processed). Since they cannot be more than s, we store them explicitly sorted (say, in increasing index) in a table B so that membership queries can be answered in $O(\log s)$ time.

Second phase. The second phase is very similar to the first one[1]: pick s sites to process, use Lemma 4.1 to find an edge for each site, once all edges of a site have been found, replace it with another site, and continue until only big sites remain. The main difference is that we now report every edge of the diagram that we compute provided that (i) it does not lie on a bisector between two big sites, and (ii) it has not been reported before. The second condition is detected as follows: suppose while scanning the cell of v_i we find an edge e that lies on the bisector of v_i and v_j. We report e only if either (iia) both v_i and v_j are small sites and $i < j$, or (iib) v_i is a small site and v_j is a big site.

Third phase. The aim of the third phase is to report every edge of the diagram that is on the bisector between two big sites. For this, we compute the Voronoi diagram of the big sites in $O(s \log s)$ time. Let E_B denote the set of its edges. The edges of E_B that are also present in the Voronoi diagram of P are the ones that need to be reported (note that possibly only a portion of these edges need to be reported).

In order to confirm which edges of E_B remain in the diagram we proceed in a similar way as the second scan of Lemma 4.1: in each step we compute the Voronoi diagram of B and a batch of s sites of P. For any edge of E_B, we check whether it is cut off in the new diagram. If so, we update its endpoints in E_B and we continue with the next batch of s sites of P. After processing all the sites of P, the remaining $O(s)$ edges in E_B that have not become empty constitute all the edges of the Voronoi diagram of P which are on a bisector of two big sites. Notice that in this procedure, in contrast to Lemma 4.1, we report $O(s)$ edges that are not necessarily incident to s different cells.

▶ **Theorem 4.2.** *Let $P = \{p_1, \ldots, p_n\}$ be a planar n-point set in general position stored in a read-only array. We can report edges of* NVD(P) *(or* FVD(P)*) in $O((n^2/s) \log s)$ time using $O(s)$ words of workspace.*

Proof. Lemma 4.1 certifies that edges reported in the second phase are part of NVD(P) (or FVD(P)). Also, conditions (iia) and (iib) make sure that no edge is reported more than once. The reasoning for edges reported in the third phase is similar. Clearly, if an edge $e \in$ NVD(P) (or $e \in$ FVD(P)) is between two big sites, the same edge (possibly a superset) must also be present in NVD(B) (or FVD(B)). The reverse inclusion follows from exhaustiveness: for each edge of NVD(B) (or FVD(B)) we consider all sites of P and for each one, we remove only the portion of the edge that is on the wrong side of the bisector.

Now we argue about running time. Computationally speaking, the most expensive part of the algorithm is in the $O(n/s)$ executions of Lemma 4.1 that are done in the first and second phases. Other than that, creating table B needs $O(s \log s)$ time, and we make $O(n)$ lookups in B, two per edge of NVD(P) (or FVD(P)). Each lookup needs $O(\log s)$ time, so $O(n \log s)$ in total. The third phase makes a single scan of the input, thus it takes $O(n \log s)$ time. For the Farthest Voronoi algorithm we also compute the vertices of the convex hull using the approach of Darwish and Elmasry [18] for which the running time is $o((n^2/s) \log s)$, so non-critical.

During the execution of the algorithm we store only s sites that are currently being processed (along with $O(1)$ information attached to each of them), the structure B of less than s big sites, the batch of s sites being processed (and its associated Voronoi diagram). All of this can be stored using $O(s)$ words of workspace, as claimed. ◀

[1] Indeed, the first and second phases are so similar that they can be merged. However, as we will see afterwards, this is not possible for higher order Voronoi diagrams. Thus, for coherence we split the phases even for the $k = 1$ case.

5 Higher-Order Voronoi Diagrams

We now consider the case of higher-order Voronoi diagrams. More precisely, we are given an integer $K \in O(\sqrt{s})$, and we would like to report the edges of all Voronoi diagrams of order up to K. For this, we generalize our approach from the previous section, and we combine it with a recursive procedure: for $k = 1, \ldots, K-1$, we compute the edges of $\mathrm{VD}^{k+1}(P)$ by using previously computed edges of $\mathrm{VD}^k(P)$. To make efficient use of available memory, we perform the computations of $\mathrm{VD}^k(P)$ in a pipelined fashion, so that in each stage, the necessary edges of the previous Voronoi diagram are available.

We begin with some preliminary remarks. We call a cell C of $\mathrm{VD}^k(P)$ a *k-cell*, and we represent it as the set of k sites that are closest to all points in C. Similarly, we call a vertex v of $\mathrm{VD}^k(P)$ a *k-vertex*, and we represent it as a set of $k+2$ sites of P, namely $k-1$ sites that are closest to v, and the three sites that come next in the distance order and are equidistant to v. Finally, the edges of $\mathrm{VD}^k(P)$ are called *k-edges*. We represent them in a somewhat unusual manner: each edge of $\mathrm{VD}^k(P)$ is split into two directed *half-edges*, such that the half-edges are oriented in opposing directions and such that each half-edge is associated with the k-cell to its left. A half-edge e is represented by $k+3$ sites of P: the $k-1$ sites closest to e, the two sites that come next in the distance order and are equidistant to e, and two more sites needed to define the vertices at the endpoints of e. The order of the vertices encodes the direction of the half-edge. The half-edge is directed from the *tail* vertex to the *head* vertex. We will need several well-known properties of higher-order Voronoi diagrams: (I) let $Q_1, Q_2 \subset P$ be two k-subsets such that the k-cells $C^k(Q_1)$ and $C^k(Q_2)$ are non-empty and adjacent (i.e., share an edge e). Then, the set $Q = Q_1 \cup Q_2$ has size $k+1$, and $C^{k+1}(Q)$ is a non-empty $(k+1)$-cell [23]. (II) Let $Q \subset P$ be a $(k+1)$-subset such that $C^{k+1}(Q)$ is non-empty. Then, the portion of $\mathrm{VD}^k(P)$ restricted to $C^{k+1}(Q)$ is identical to (i.e., has the same vertices and edges as) the portion of $\mathrm{FVD}(Q)$ restricted to $C^{k+1}(Q)$. Furthermore, the edges of $\mathrm{FVD}(Q)$ in $C^{k+1}(Q)$ do not intersect the boundary, but their endpoints either lie in the interior of $C^{k+1}(Q)$ or coincide with vertices of $C^{k+1}(Q)$. Hence, every $(k+1)$-cell contains at most $O(k+1)$ k-edges and at least one k-edge, and these edges form a tree [23]. (III) Every k-vertex is either also a $(k-1)$-vertex or also a $(k+1)$-vertex. In particular, every vertex appears in exactly two Voronoi diagrams of consecutive order. We call a k-vertex *old*, if it is also a $(k-1)$-vertex, and *new* otherwise. (All 1-vertices are new).

Next, we describe a procedure to generate all (directed) $(k+1)$-half-edges, assuming that we have the k-half-edges at hand. Later, we will combine these procedures in a space-efficient manner. At a high level, our idea is as follows: let e be a k-half-edge. By property (II), the half-edge e lies inside a $(k+1)$-cell C. We will see that we can use e as a starting point to report all half-edges of C, similarly as in Lemma 4.1. However, if we repeat this procedure for every k-half-edge, we may report a $(k+1)$-half-edge $\Omega(k)$ times. This will lead to problems when we combine the algorithms for computing the different orders. To avoid this, we do the following. We call a k-half-edge *relevant* if its tail vertex lies on the boundary of the $(k+1)$-cell that contains it. For each $(k+1)$-cell C, we partition the boundary of C into *intervals* of $(k+1)$-half-edges that lie between two consecutive tail vertices of relevant k-half-edges. We assign each interval to the relevant k-half-edge of its clockwise endpoint. Now, our algorithm goes through all k-half-edges. If the current k-half-edge e is not relevant, the algorithm does nothing. Otherwise, it reports the $(k+1)$-half-edges of the interval assigned to e. This ensures that every half-edge is reported exactly once. As in the previous section, we distinguish between *big* and *small* cells in $\mathrm{VD}^{k+1}(P)$, lest we spend too much time on cells with many incident edges. A more detailed description follows below.

The following lemma describes an algorithm that takes s k-half-edges and for each of them either determines that it is not relevant or finds the first edge of the interval of $(k+1)$-half-edges assigned to it.

▶ **Lemma 5.1.** *Suppose we are given s different k-half-edges $e_1^k, \ldots, e_s^k$ represented by the subsets $E_1, \ldots, E_s$ of P. There is an algorithm that, for $i = 1, \ldots, s$, either determines that e_i^k is not relevant, or finds e_i^{k+1}, the first $(k+1)$-edge of the interval assigned to e_i^k. The algorithm takes total time $O(nk \log^{1+\varepsilon} k \cdot (\log s/\log k)^{O(1)})$ and uses $O(ks)$ words of workspace.*

Proof. Our algorithm proceeds analogously to Lemma 4.1. First, we inspect all k-half-edges e_i^k. If the additional site defining the tail vertex of e_i^k is one of the $k+1$ sites defining e_i^k (i.e., the sites which are closest in the distance order to e_i^k), then the tail vertex of e_i^k lies in the interior of a $(k+1)$-cell, and the edge is not relevant. Otherwise, the tail vertex will be an old $(k+1)$-vertex, and we need to determine the first $(k+1)$-half-edge for the interval assigned to e_i^k. Let I be the set of all indices i such that e_i^k is relevant.

To determine the first half-edge of each interval, we process the sites in P in batches of size s. In each iteration, we pick a new batch Q of s sites. Then, we construct $\mathrm{VD}^{k+1}\big(\bigcup_{i\in I} E_i \cup Q\big)$ in $O(sk \log^{1+\varepsilon} k \cdot (\log s/\log k)^{O(1)})$ time [15]. By construction, the tail vertex of each e_i^k with $i \in I$ belongs to the resulting diagram. Thus, we iterate over all batches, and for each e_i^k, we determine the edge f_i^{k+1} that appears in one of the resulting diagrams such that (i) f_i^{k+1} is incident to the tail vertex of e_i^k; (ii) f_i^{k+1} is to the right of the directed line spanned by e_i^k; and (iii) among all such edges, f_i^{k+1} makes the smallest angle with e_i^k. We need $O(n/s)$ iterations to find f_i^{k+1}. Now, for each $i \in I$, the desired $(k+1)$-half-edge e_i^{k+1} is a subset of f_i^{k+1}. Thus, as in Lemma 4.1, we perform a second scan over P to find the other endpoint of e_i^{k+1}. We orient e_i^{k+1} such that the cell containing e_i^k lies to the left of it.

It follows that we can process s edges of $\mathrm{VD}^k(P)$ in $O(nk \log^{1+\varepsilon} k \cdot (\log s/\log k)^{O(1)})$ time using a workspace with $O(ks)$ words to store the s different subsets for the edges. ◀

The algorithm from Lemma 5.1 is actually more general. If, instead of a k-half-edge e_i^k that lies inside a $(k+1)$-cell C, we have a $(k+1)$-half-edge e_i^{k+1} that lies on the boundary of C, the same method of processing P in batches of size s allows us to find the next $(k+1)$-half-edge incident to C in counterclockwise order from e_i^{k+1}. These two kinds of edges can be handled simultaneously.

▶ **Corollary 5.2.** *Suppose we are given s half-edges $e_1, \ldots, e_s$ such that each e_i is either a k-half-edge or a $(k+1)$-half-edge. Then, we can find in total time $O(nk \log^{1+\varepsilon} k \cdot (\log s/\log k)^{O(1)})$ and using $O(ks)$ words of workspace a sequence $f_1, \ldots, f_s$ of $(k+1)$-half-edges such that, for $i = 1, \ldots, s$, we have*

1. *if e_i is a relevant k-half-edge, then f_i is the first $(k+1)$-half-edge of the interval for e_i;*
2. *if e_i is k-half-edge-that is not relevant, then $f_i = \perp$; or*
3. *if e_i is a $(k+1)$-half-edge, then f_i is the counterclockwise successor of e_i.*

▶ **Lemma 5.3.** *Using two scans over all k-half-edges, we can report the $(k+1)$-half-edges in batches of size at most s such that each $(k+1)$-half-edge is reported exactly once. This takes $O\big(\frac{n^2k^2}{s} \log^{1+\varepsilon} k \cdot (\log s/\log k)^{O(1)}\big)$ time using $O(ks)$ words of workspace.*

Proof. The algorithm consists of three phases: In the first phase, we keep s half-edges $e_1, \ldots, e_s$ such that each e_i is either a k-half-edge or a $(k+1)$-half-edge. In each iteration, we apply Corollary 5.2 to these half-edges, to obtain s new $(k+1)$-half-edges $f_1, \ldots, f_s$. Now, for each $i = 1, \ldots, s$, three cases apply: (i) $f_i = \perp$, i.e., e_i was not relevant. In the next iteration,

we replace e_i with a fresh k-half-edge; (ii) $f_i \neq \perp$. Now we need to determine whether f_i is the last half-edge of its interval. For this, we check whether the head vertex of f_i is a new $(k+1)$-vertex. As in Lemma 5.1, this can be done by checking whether the additional site that determines the head vertex of f_i is among the $k+2$ sites determining f_i. If f_i is not the last half-edge of its interval, we set e_i to f_i for the next iteration; otherwise, we set e_i to a fresh k-half-edge. We repeat this procedure until there are no fresh k-half-edges left.

The remaining k-half-edges in the working memory are incident to the big $(k+1)$-cells. We store them in an array B^{k+1}, sorted according to the lexicographic order of the indices of their sites. We emphasize that in the first phase, we do not report any $(k+1)$-edge.

In the second phase, we repeat the same procedure as in the first phase. However, this time we report (i) every $(k+1)$-half-edge incident to a small $(k+1)$-cell; and (ii) the opposite direction of every $(k+1)$-half-edge e incident to a small $(k+1)$-cell, so that the $(k+1)$-cell on the right of e is a big $(k+1)$-cell. We use B^{k+1} to identify the big cells.

In the third phase we report every $(k+1)$-half-edge e that is incident to a big $(k+1)$-cell, while the $(k+1)$-cell on the right of e is also a big $(k+1)$-cell. Let $\{B^{k+1}\}$ denote the sites that define the big $(k+1)$-cells. We construct $\mathrm{VD}^{k+1}(\{B^{k+1}\})$ in the working memory. Then, we go through the sites in P in batches of size s, adding the sites of each batch to $\mathrm{VD}^{k+1}(\{B^{k+1}\})$. While doing this, as in the algorithm for Lemma 4.2, we keep track of how the edges of $\mathrm{VD}^{k+1}(\{B^{k+1}\})$ are cut by the new diagrams. In the end, we report all $(k+1)$-edges of $\mathrm{VD}^{k+1}(\{B^{k+1}\})$ that are not empty. By *report*, we mean report two $(k+1)$-half-edges in opposing directions. As we explained in the algorithm for Lemma 4.2, these $(k+1)$-half-edges cover all the $(k+1)$-half-edges incident to a big $(k+1)$-cell, while their right cell is also a big $(k+1)$-cell.

Regarding the running time, the first and the second phase consists of $O(nk/s)$ applications of Corollary 5.2 which takes $O\big(\frac{n^2k^2}{s}\log^{1+\varepsilon}k\cdot(\log s/\log k)^{O(1)}\big)$ time. Sorting the big $(k+1)$-cells in B^{k+1} takes $O(ks(\log k+\log s))$ steps: we sort the indices of the sites of each big $(k+1)$-cell in $O(k\log k)$ steps. Then we sort the big cells, where each comparison in the lexicographic order requires $O(k)$ steps, for a total of $O(ks\log s)$ steps.

A query in B^{k+1} takes $O(k\log k+\log s)$ time: given a query $(k+1)$-cell C we sort its indices in $O(k\log k)$ time. Then we use binary-search to find cells in B^{k+1} with the same first index as C. Among these, we continue the binary-search with comparing the second indices, and so on. Thus, in each step we compare only one index of two $(k+1)$-cells, and either the size of search set is halved, or the search continues with the next index of C. Thus, searching a cell C with sorted index in B^{k+1} requires $O(k+\log s)$ time.

The algorithm performs at most two searches in B^{k+1} per each $(k+1)$-half-edge, for a total of $O(nk)$ edges. In the third phase, constructing a $(k+1)$-order Voronoi diagram of $O(ks)$ sites takes $O(sk\log^{1+\varepsilon}k\cdot(\log s/\log k)^{O(1)})$ time. We repeat it $O(n/s)$ times, which takes $O(nk\log^{1+\varepsilon}k\cdot(\log s/\log k)^{O(1)})$ time in total.

Overall, the running time of the algorithm simplifies to $O\big(\frac{n^2k^2}{s}\log^{1+\varepsilon}k\cdot(\log s/\log k)^{O(1)}\big)$. The algorithm uses a workspace of $O(sk)$ words, for running Corollary 5.2, for storing big $(k+1)$-cells and for constructing Voronoi diagrams of size $O(ks)$. ◀

Now, in order to find k-half-edges for all $k=1,\ldots,K$, we proceed as follows: first, we compute s 1-edges (notice that we report every 1-edge as two 1-half-edges in opposing directions). Then, we apply Lemma 5.3 in a pipelined fashion to obtain k-half-edges for all $k=2,\ldots,K$. In each iteration, the algorithm from Lemma 5.3 consumes at most s k-half-edges from the previous order and produces at most $2s$ $(k+1)$-half-edges to be used at the next order. This means that if we have between s and $3s$ new k-half-edges available in a buffer, then we can use them one by one whenever the algorithm for computing k-half-edges

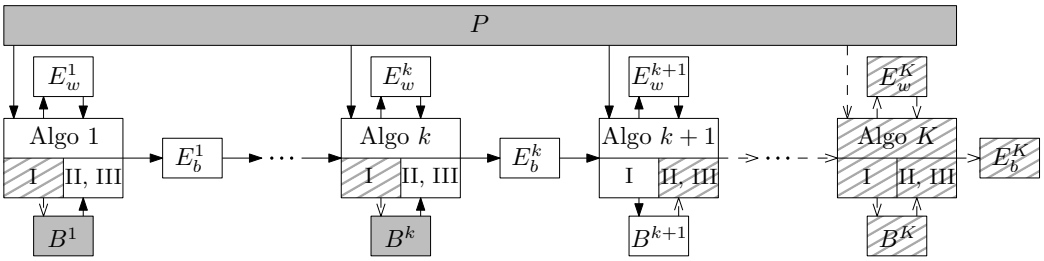

Figure 2 This diagram shows the algorithm in *stage* k. For $i = 1, \ldots, K$, Algo i is the algorithm for computing i-edges. The roman numerals I, II and III refer, respectively, to the first, second and third phase of Algo i. The white and the tiled rectangles are, respectively, active and inactive parts in stage k of the main algorithm. E_w^i, E_b^i and B^i indicate memory cells, and they are, respectively, the working memory of Algo i, the buffer for i-edges, and the big i-cells. The algorithm in stage k, does not use the tiled memory cells, and it uses the gray ones only for reading the data that has been produced in previous stages. The arrows show reading from or writing to memory cells, and the dashed arrows are inactive in stage k. In stage k, all the k-half-edges are reported and the big $(k+1)$-cells are inserted into B^{k+1} for being used in the next stages.

in Lemma 5.3 requires such a new k-half-edge. Whenever a buffer falls below s half-edges, we run the algorithm for the previous order until the buffer size is again between s and $3s$. Applying this idea for all the orders $k = 1, \ldots, K-1$, we need to store $K-1$ buffers, each containing up to $3s$ half-edges for the corresponding diagram. Furthermore, for each diagram, we need to store the current workspace required by the algorithm to produce new edges (as in Lemma 5.3). Since a k-edge is represented by $O(k)$ sites from P, the buffer for k-edges requires $O(ks)$ words of memory. We denote it by E_b^k. Furthermore, by Lemma 5.3, the new k-edges can be found using $O(ks)$ words of workspace, which we denote by E_w^k.

▶ **Theorem 5.4.** *Let $K \in O(\sqrt{s})$ and $P = \{p_1, \ldots, p_n\}$ be a planar n-point set in general position, given in a read-only array. We can report all the edges of $\mathrm{VD}^1(P), \ldots, \mathrm{VD}^K(P)$ in $O\big(\frac{n^2K^6}{s} \log^{1+\varepsilon} K \cdot (\log s / \log K)^{O(1)}\big)$ time using a workspace of size $O(s)$.*

Proof. We compute the half-edges of $\mathrm{VD}^1(P), \ldots, \mathrm{VD}^K(P)$ simultaneously, in a pipelined fashion. For $k = 1$ we use the algorithm of Theorem 4.2 and for $k = 2, \ldots, K$, we run the algorithm from Lemma 5.3 to compute k-edges. The algorithm for computing $\mathrm{VD}^k(P)$, has its own working memory, denoted by E_w^k and an output buffer E_b^k. In addition, it has an array B^k to store the big k-cells for being used in the second and third phase of the algorithm. Each of these arrays should be able to store $O(s')$ half-edges and cells of $\mathrm{VD}^k(P)$, for $s' = s/K^2$. Since we need $O(k)$ sites to represent a k-half-edge or a k-cell, the total space requirement for all algorithms is $O(s)$.

We now describe how the simultaneous algorithms interact. Our algorithm works in *stages*. In stage 0, we perform only the first phase of Theorem 4.2, to find the $O(s')$ big cells of $\mathrm{VD}^1(P)$, and we store them in B^1. Now we know the big 1-cells. Then, in stage 1, we perform the second phase of Theorem 4.2 to find and report the half-edges of $\mathrm{VD}^1(P)$ in batches of size at most $2s'$, and we store these 1-half-edges in E_b^1. Whenever we have at least s' half-edges in E_b^1, we pause the algorithm of Theorem 4.2, and we perform the first phase of Lemma 5.3 to find half-edges of $\mathrm{VD}^2(P)$ with E_b^1 as input. Whenever the algorithm for $\mathrm{VD}^2(P)$ requires new 1-half-edges, and the buffer E_b^1 falls below s' half-edges, we continue running the algorithm for $\mathrm{VD}^1(P)$. When the algorithm for $\mathrm{VD}^2(P)$ has consumed all 1-half-edges and there are less than s' half-edges in E_w^2, then we stop the algorithm for $\mathrm{VD}^2(P)$. The half-edges in E_w^2 represent the big cells of $\mathrm{VD}^2(P)$, and we store them in B^2. This concludes stage 1.

In general, in stage k of the algorithm, we identified the big cells $B^1, \ldots, B^k$ of the first k diagrams. We perform the second and the third phase of Theorem 4.2 and Lemma 5.3 in a pipelined fashion to generate the half-edges of $\mathrm{VD}^1(P), \ldots, \mathrm{VD}^k(P)$ and store them in the buffers $E_b^1, \ldots, E_b^k$. We report the half-edges of $\mathrm{VD}^k(P)$ and if $k \neq K$, we also use E_b^k as an input of the first phase of Lemma 5.3, which gives us B^{k+1} for the next stage; see Figure 2. Using this procedure, we report every half-edge of $\mathrm{VD}^1(P), \ldots, \mathrm{VD}^K(P)$ exactly once.

Regarding the running time, in each stage $k = 1, \ldots, K$, we have to compute all diagrams $\mathrm{VD}^1(P), \ldots, \mathrm{VD}^k(P)$, using Lemma 5.3. This takes $O\big(\frac{n^2k^3}{s'} \log^{1+\varepsilon} k \cdot (\log s'/\log k)^{O(1)}\big)$ time. The running time for stage 0 is negligible. The complete algorithm takes $O\big(\frac{n^2K^4}{s'} \log^{1+\varepsilon} K \cdot (\log s'/\log K)^{O(1)}\big)$ time, which is $O\big(\frac{n^2K^6}{s} \log^{1+\varepsilon} K \cdot (\log s/\log K)^{O(1)}\big)$, in terms of s. ◀

Acknowledgements. The authors would like to thank Luis Barba, Kolja Junginger, Elena Khramtcova, and Evanthia Papadopoulou for fruitful discussions on this topic.

References

1 Alok Aggarwal, Leonidas J. Guibas, James B. Saxe, and Peter W. Shor. A linear-time algorithm for computing the voronoi diagram of a convex polygon. *Discrete Comput. Geom.*, 4:591–604, 1989.

2 Boris Aronov, Matias Korman, Simon Pratt, André van Renssen, and Marcel Roeloffzen. Time-space trade-offs for triangulating a simple polygon. In *Proc. 15th Scand. Sympos. Workshops Algorithm Theory (SWAT)*, volume 53, pages 30:1–30:12, 2016.

3 Sanjeev Arora and Boaz Barak. *Computational Complexity.A modern approach*. Cambridge University Press, 2009.

4 Tetsuo Asano, Kevin Buchin, Maike Buchin, Matias Korman, Wolfgang Mulzer, Günter Rote, and André Schulz. Memory-constrained algorithms for simple polygons. *Comput. Geom.*, 46(8):959–969, 2013.

5 Tetsuo Asano and David Kirkpatrick. Time-space tradeoffs for all-nearest-larger-neighbors problems. In *Proc. 13th Int. Symp. Algorithms and Data Structures (WADS)*, pages 61–72, 2013.

6 Tetsuo Asano, Wolfgang Mulzer, Günter Rote, and Yajun Wang. Constant-work-space algorithms for geometric problems. *J. of Comput. Geom.*, 2(1):46–68, 2011.

7 Tetsuo Asano, Wolfgang Mulzer, and Yajun Wang. Constant-work-space algorithms for shortest paths in trees and simple polygons. *J. Graph Algorithms Appl.*, 15(5):569–586, 2011.

8 Franz Aurenhammer, Rolf Klein, and Der-Tsai Lee. *Voronoi diagrams and Delaunay triangulations*. World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ, 2013.

9 Yeganeh Bahoo, Bahareh Banyassady, Prosenjit Bose, Stephane Durocher, and Wolfgang Mulzer. Finding the k-visibility region of a point in a simple polygon in the memory-constrained model. In *Proc. 32nd European Workshop Comput. Geom. (EWCG)*, 2016.

10 Luis Barba, Matias Korman, Stefan Langerman, Kunihiko Sadakane, and Rodrigo I. Silveira. Space–time trade-offs for stack-based algorithms. *Algorithmica*, 72(4):1097–1129, 2015.

11 Luis Barba, Matias Korman, Stefan Langerman, and Rodrigo I. Silveira. Computing the visibility polygon using few variables. *Comput. Geom.*, 47(9):918–926, 2013.

12 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry. Algorithms and applications*. Springer-Verlag, third edition, 2008.

13 Allan Borodin and Stephen A. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11:287–297, 1982.

14 Hervé Brönnimann, Timothy M. Chan, and Eric Y. Chen. Towards in-place geometric algorithms and data structures. In *Proc. 20th Annu. Sympos. Comput. Geom. (SoCG)*, pages 239–246, 2004.

15 Timothy M Chan. Remarks on k-level algorithms in the plane. *manuscript, Univ. of Waterloo*, 1999.

16 Timothy M Chan. Random sampling, halfspace range reporting, and construction of $(\leq k)$-levels in three dimensions. *SIAM J. Comput.*, 30(2):561–575, 2000.

17 Timothy M. Chan and Eric Y. Chen. Multi-pass geometric algorithms. *Discrete Comput. Geom.*, 37(1):79–102, 2007.

18 Omar Darwish and Amr Elmasry. Optimal time-space tradeoff for the 2D convex-hull problem. In *Proc. 22nd Annu. European Sympos. Algorithms (ESA)*, pages 284–295, 2014.

19 Amr Elmasry and Frank Kammer. Space-efficient plane-sweep algorithms. `arXiv : 1507.01767`, 2015.

20 Sariel Har-Peled. Shortest path in a polygon using sublinear space. *J. of Comput. Geom.*, 7(2):19–45, 2016.

21 Matias Korman. Memory-constrained algorithms. In *Encyclopedia of Algorithms*, pages 1260–1264. Springer Berlin Heidelberg, 2016. `doi:10.1007/978-3-642-27848-8_586-1`.

22 Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Time-space trade-offs for triangulations and Voronoi diagrams. In *Proc. 14th Int. Symp. Algorithms and Data Structures (WADS)*, pages 482–494, 2015.

23 Der-Tsai Lee. On k-nearest neighbor Voronoi diagrams in the plane. *IEEE Trans. Computers*, 31(6):478–487, 1982.

24 J. Ian Munro and Mike Paterson. Selection and sorting with limited storage. *Theoret. Comput. Sci.*, 12:315–323, 1980.

25 J. Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. *Theoret. Comput. Sci.*, 165(2):311–323, 1996.

26 J. Pagter and T. Rauhe. Optimal time-space trade-offs for sorting. In *Proc. 39th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 264–268, 1998.

27 Ira Pohl. A minimum storage algorithm for computing the median. Technical Report RC2701, IBM, 1969.

28 John E. Savage. *Models of computation – exploring the power of computing.* Addison-Wesley, 1998.